A DSL based toolchain for design space exploration in structured parallel programming

Danelutto, M., Torquati, M., & Kilpatrick, P. (2016). A DSL based toolchain for design space exploration in structured parallel programming. *Procedia Computer Science* , *80*, 1519-1530. https://doi.org/10.1016/j.procs.2016.05.477

Published in:
Procedia Computer Science

Document Version:
Publisher's PDF, also known as Version of record

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Procedia Computer Science

Volume 80, 2016, Pages 1519–1530

ICCS 2016. The International Conference on Computational Science

A DSL based toolchain for design space exploration in structured parallel programming

Marco Danelutto[1], Massimo Torquati[1], and Peter Kilpatrick[2]

[1] Dept. Computer Science, Univ. of Pisa, Italy
{marcod,torquati}@cse.concordia.ca
[2] Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk

Abstract

We introduce a DSL based toolchain supporting the design of parallel applications where parallelism is structured after *parallel design pattern* compositions. A DSL provides the possibility to write high level parallel design pattern expressions representing the structure of parallel applications, to refactor the pattern expressions, to evaluate their non-functional properties (e.g. ideal performance, total parallelism degree, etc.) and finally to generate parallel code ready to be compiled and run on different target architectures. We discuss a proof-of-concept prototype implementation of the proposed toolchain generating FastFlow code and show some preliminary results achieved using the prototype implementation.

Keywords: parallel design patterns, DSL, design space exploration, non-functional concerns

1 Introduction

In the scenario considered here, an application programmer has to write a parallel application solving a given problem. He/she knows the target architecture on which the application should be executed and can use any of several parallel programming frameworks supporting a general purpose, comprehensive set of high level parallel design patterns [12, 5]. Ideally, the application programmer may orchestrate the parallel structure of the application in various ways, using different combinations (nestings) of available high level parallel patterns, and/or choosing different non-functional parameters for a single parallel pattern composition (e.g. parallelism degree or task scheduling policy). The corresponding tuning activities may be regarded as the *exploration* of a space of admissible and functionally equivalent solutions (with different non-functional properties) that may all be logically derived from an original parallel design pattern expression modelling the application at hand. For example, a programmer facing the parallel implementation of video surveillance system may identify that each of the video frames should first be prepared for processing (surveillance cameras often work in extreme lighting conditions) and then differences w.r.t. previous images in the stream have to be detected and evaluated.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.477

```
<patt>     ::= <seqp> | <stream> | <dataparp>
<seqp>     ::= Seq(<funname>) | SeqComp(<patt>,<patt>) | StreamSource(<funname>) |
               StreamSink(<funname>) |DataSource(<funname>) | DataSink(<funname>)
<streamp>  ::= Pipeline(<patt>,<patt>) | Farm(<patt>)
<dataparp> ::= Map(<pat>) | Reduce<Pat>
```

Figure 1: Sample pattern expression grammar

The parallel structure of the application may be orchestrated as a pipeline of stages (image cleaning, motion detection, evaluation) or as a stream map whose parallel "worker" activities compute all the three phases on a single frame of the video stream. These alternative designs, while functionally equivalent, may exhibit notably different non-functional behaviour.

The exploration of the space solution is not an easy task, *per se*. First, deriving alternative, functionally equivalent application parallel structures expressed in terms of the available high level parallel design patterns requires significant parallel programming ability. Second, determining which non-functional design pattern parameter values are most suitable is also a difficult task. For example, identifying the "optimal" parallelism degree, requires quite deep knowledge of the target architecture features *and* of the computational grain of the parallel pattern components. In addition, setting of the values for one of the non-functional aspects is influenced and influences the values chosen for the other non-functional parameters, which makes the optimization process "hard". Last but not least, each solution in the space needs considerable coding activity to produce a running application. Depending on the parallel programming framework, the coding activity may consist in significantly different effort, and very often debugging and tuning of the resulting code is a cumbersome and lengthy process.

Our goal here is to introduce and discuss a DSL based tool suitable for supporting experimentation with different, alternative and functionally equivalent, parallel implementations of an application at a *high level of abstraction*, *before* moving to the actual coding phase. Moreover, the tool we propose will eventually support parallel code production directly from the high level parallel application structure expressed in terms of parallel design pattern compositions.

The contribution of the paper consists in: 1) a parallel design pattern based DSL suitable for expressing a large number of parallel applications; 2) a formal system supporting the exploration of alternative implementations of a given parallel application, along with evaluation of the actual behaviour, in terms of non-functional properties; 3) a compiler generating parallel application code in a well-known parallel programming framework directly from the high level DSL representation of the parallel application.

The paper is organized as follows: Sec. 2 introduces the DSL, Sec. 3 discusses the alternative usages of the DSL framework, Sec. 4 presents some preliminary experimental results and Sec. 5 discusses the related work. Finally Sec. 6 draws conclusions and outlines future work.

2 Parallel DSL toolchain

The DSL we propose is fundamentally a high level pattern grammar supporting annotations and typical Abstract Syntax Tree (AST) processing patterns: visitors, refactorers, (partial) evaluators. In this paper, we will assume that a limited number of parallel patterns is used. We will use those patterns that are universally recognized as valid, common and useful parallel patterns. Ideally, the pattern set considered may be extended in any direction–e.g. including domain specific or higher level patterns–provided the principles relative to the DSL design and

usage are fulfilled. Besides "classical" parallel patterns such as those listed in [12], we will consider also several notable patterns as identified in the algorithmic skeleton research area [10].

2.1 The building block grammar

Our core DSL pattern grammar is outlined in Fig. 1. All the patterns in the set are well-known parallel patterns, available in a number of pattern/skeleton programming frameworks (such as [9, 8, 5]). The sequential patterns include wrapping of sequential portions of code (functions) and portions of code producing a stream of items from external (e.g. frames from a video camera or network packets from a network interface) or internal sources (e.g. data items coming from storage or from main memory) and consuming a stream of input items (stream source and sink, respectively). Stream parallel patterns include pipeline (computations in stages) and farm (independent computation of the same function over all the stream items). Data parallel patterns include map (independent computation of the same function over all the items of a data collection) and reduce (summing up all items in a data collection by means of an associative operator).

Despite the fact that the pattern set represented in the grammar is quite limited, a wide range of applications may be modelled using the DSL, such as, for example:

- video stream processing[1]: Pipeline(StreamSource(Reader),Seq(filter),StreamSink(Writer))
- mapreduce applications: SeqComp(Map(...),Reduce(...)) or Pipeline(Map(...),Reduce(...))
- applications running data parallel computations on each item of an input stream: Farm(Map(...)).

There are well-known functional equivalences that can be expressed in terms of the DSL grammar, such as those listed in Fig. 2[2]. As an example, the rule $\mathsf{Farm}(\mathsf{Pattern}_1) \equiv \mathsf{Pattern}_1$ states that a stream parallel computation applying a given computation (denoted by $\mathsf{Pattern}_1$) over all the items of a stream *in parallel*, is equivalent to applying to all the items of the stream the same computation *sequentially*. The *map fusion* rule states that applying two functions in a composition of maps is equivalent to applying the sequential composition of the two functions in two distinct maps, instead. The application of these rules enables the programmer to produce pattern terms in a *patterned application space* which are functionally equivalent, and thus worthy of consideration before starting to code the application.

2.2 Non-functional property management

A number of non-functional concerns, such as performance, efficiency, power consumption, security, etc., should be taken into account in order to determine which application structure (in terms of patterns) is the "best" one to implement on a given target architecture. Having identified a number of different functionally correct application structures, we need to evaluate them according to non-functional properties to determine which is the most appropriate. In particular, we wish to establish mechanisms and tools such that we can compute non-functional properties of a pattern expression in terms of the non-functional properties of its components.

[1] we assume here that Pipeline(A,B,C) is actually a Pipeline(Pipeline(A,B),C)

[2] for the sake of simplicity, throughout the paper we refer only to *stateless* pattern components, that is we assume any <SeqPattern > is purely functional code. When stateful patterns are considered [6] the models turn out to be more complex.

Rule name	Rule
Farm intro/elim	Farm(Pat_1) ≡ Pat_1
Pipe intro/elim	Pipeline(Pat_1, Pat_2) ≡ SeqComp(Pat_1, Pat_2)
Map fusion	SeqComp(Map(Pat_1),Map(Pat_2)) ≡ Map(SeqComp(Pat_1, Pat_2))
Reduce promotion	Pipeline(Map(Pat_1), Reduce(Pat_2)) ≡ Pipeline(Map(SeqComp(Pat_1,Reduce(Pat_2))),Reduce(Pat_2))
Data <>stream	Map(Pat) ≡ Pipeline(StreamSource,Farm(Pat),StreamSink)

Figure 2: Sample pattern equivalence rules

2.2.1 Annotations

We assume that each non-functional concern of interest for parallel programming is based on different *properties*. In turn, properties may have values that may be associated to the different DSL grammar terms. Property values may be primitive, synthesized or inherited. Primitive property values are provided through ground attributes (functor with an associated value) of grammar terms. Synthesized values may be computed for a given grammar term from the property values associated with the term's components. Inherited values may be inferred from the same property values associated to the root/ancestor components.

The extended DSL grammar including annotations may thus be defined as follows:

⟨*annotation*⟩ ::= ⟨*functor*⟩(⟨*value*⟩)
| Synthesized(⟨*annotation*⟩)
| Inherited(⟨*annotation*⟩)
| ⟨*annotation*⟩ and ⟨*annotation*⟩

⟨*functor*⟩ ::= ⟨*identifier*⟩

⟨*annotated-pattern*⟩ ::= ⟨*pattern*⟩
| ⟨*pattern*⟩ with ⟨*annotation*⟩

As an example, assume the non-functional concern of interest is performance. Pattern performance is usually a function of the performance of its component parts and primitive components (e.g. the sequential pattern components) having ground performance values. In a Pipeline(Seq(f), Seq(g)) pattern, the performances of the sequential components are ground values (e.g. service times or latencies), while the performance of the pipeline may be expressed in terms of the performances of its components/stages: the service time is the maximum of the service time of the pipeline components and its latency is the sum of the component latencies.

As a further example, assume the non-functional concern of interest is security. Security of a pattern term may be required which implies that the component parts of the patterns must be themselves secure. Therefore security attributes may be assigned as inherited annotations.

In the following, we will use Ts and Pd as functors for the annotations of service time and parallelism degree, respectively.

2.2.2 Visitors

A DSL grammar term visitor is a procedure computing the value of a property of a given non-functional concern associated to a generic grammar term. Synthesized property values may be computed with a bottom-up visitor, while inherited ones are computed by top-down visitors.

When dealing with synthesized annotations, we start with terms that have ground annotations. Ground term annotations are typically provided by the DSL user or automatically derived by a tool. If the term does not have an associated ground term annotation relative to the property, then a synthesized property value is computed by the visitor pattern according to a model of the non-functional concern at hand. When dealing with inherited annotations, we proceed the other way round.

Service time		ParDegree	
Seq( ) with Ts(x)	x	Seq( )	1
Pipeline(X,Y)	max { ServiceTime(X), ServiceTime(y) }	Farm(X)	Pd(X) × Pd(Farm(X)
Farm(X)	ServiceTime(X) / Pd(Farm(X))	Pipeline(X,Y)	Pd(X) + Pd(Y)

Figure 3: Sample synthesized attribute models (for bottom-up visitors)

We start with a ground annotated root node and propagate the inherited annotations to the node parts, again according to a model for the non-functional concern at hand.

As an example, consider again the Pipeline(Seq(f), Seq(g)) pattern expression. We assume that we have ground service time annotations for both sequential components and no annotation for the Pipe term (say Seq(f) with Ts(6.0sec) and Seq(g) with Ts(3.0sec)).

The performance service time visitor will therefore compute a bottom-up visit of the grammar term at hand, determining service time for the pipe according to the formula Ts(Pipeline(X,Y)) = max { Ts(X), Ts(Y)} and so it will eventually produce an annotated term such as Pipeline(Seq(f) with Ts(6.0sec), Seq(g) with Ts(3.0sec)) with synthesized(Ts(6.0sec)). Fig 3 outlines some simplified visitor algorithms relative to performance and parallelism degree for the patterns in our initial pattern set.

As a further example, assume we have a pattern term such as Pipeline(Seq(f), Farm(Seq(g))) and the application programmer requires that the second stage of the top level pipeline should be implemented in a "secure" way (i.e. having data ciphered). The CipheredData(true) annotation associated with the second stage node should be propagated, unchanged, to all the component nodes. In this case, a visitor managing the security annotations visiting the term Pipeline(Seq(f),Farm(Seq(g)) with CipheredData(true)) will eventually produce the term Pipeline(Seq(f),Farm(Seq(g) with inherited(CipheredData(true))) with CypheredData(true)))

2.2.3 Refactorers

A DSL grammar refactorer applies a rewrite rule (e.g. one of the rules listed in Table 2) to a DSL grammar term producing a functionally equivalent grammar term. The process is similar to that adopted in [3], but for the annotation usage. If the initial grammar term has associated annotations the ground annotations are kept in the rewritten term, while synthesized ones are cancelled (they need to be re-computed in the new context).

Consider again the output of the previous annotated pipeline term described in the previous section Pipeline(Seq(f) with Ts(6.0sec), Seq(g) with Ts(3.0sec)) with synthesized(Ts(6.0sec)) and assume we wish to apply the farm introduction rule on the first pipeline component. In this case we get a Pipeline(Farm(Seq(f) with Ts(6.0sec)), Seq(g) with Ts(3.0sec)). We may apply again the performance service time visitor to evince the service time for the new pattern expression. However, the service time visitor needs the parallelism degree of a farm in order to be able to compute the farm service time (see Table 3). If we associate the parallelism degree to the farm: Pipeline(Farm(Seq(f) with Ts(6.0sec) and Pd(3)), Seq(g) with Ts(3.0sec)) then the performance service time visitor will return Pipeline(Farm(Seq(f) with Ts(6.0sec)) with Pd(3), Seq(g) with Ts(2.0sec)) with Ts(3.0sec).

2.2.4 Optimizers

When a strategy is known for optimizing a given property $\mathcal{P}$ of a non-functional concern, a DSL optimizer may be provided that rewrites a grammar term T_i into a functionally equivalent

grammar term T_e such that the $\mathcal{P}$ visitor gives a better result for T_e than for T_i and T_e derives from T_i by means of a finite set of rewritings performed according to the rules of Table 2.

For example, "normal form" is a well-known optimizing strategy for stream parallel pattern compositions [2]. The DSL toolchain will therefore provide a NormalForm optimizer (multiple step rewriting rule) computing the normal form of any pattern expression.

As a second example, an optimizer may be built that transforms a stream parallel pattern expression in such a way that a given Service Level Agreement (SLA) for the service time is ensured. First the application programmer annotates the top level pattern with the SLA hosting the required service time. Then a top-down visitor recursively assigns SLAs to parts of the already annotated nodes. Sub-SLAs for pattern components are derived fully respecting the pattern semantics. Last, a bottom-up visitor ensures that the local SLA is satisfied, possibly by parallelizing the local sub-tree through application of one of the rules in Table. 2.

2.2.5 Target architecture dependencies

As described so far, some entities are "architecture (hw & sw) agnostic" while others are not. The pattern grammar with its composition rules, the visitors and the refactorer are architecture agnostic, while the optimizer and the models used to support visitors are not.

2.3 Code generation

The last component in the DSL toolchain is code generation. Starting from the DSL grammar terms, the toolchain provides the possibility to generate actual FastFlow code, FastFlow being the targeted algorithmic skeleton programming framework [5]. Different kinds of code may be generated depending on the information we have associated to the pattern expression.

In the simplest case, where not enough additional information is associated to the pattern expression, a generic ("template") code may be produced, whose business logic parts must be subsequently provided by the application programmer. In this case the application programmer is responsible for inserting into the automatically generated template code a) the data types used for the different functions and b) the actual code of the functions wrapped in Seq wrappers.

If enough information (possibly automatically extracted from existing business logic code) exists, such that the "funname" parameter of the Seq() pattern allows precise identification of the exact body of the function along with the data types used for input and output parameters, then complete FastFlow application code may be generated, ready for linking with the (source or library) files hosting the sequential business logic code.

3 DSL tool usage

Let us suppose we have a complete pattern grammar, a full set of models for all the non-functional properties of interest, with corresponding visitors and optimizer. What is the typical usage of the DSL toolchain? We can envisage two different patterned application design "methodologies": one providing the application programmer complete freedom in exploring the solution space before producing code, and the other one more reliant on optimizers to derive suitable code for the target architecture at hand.

The starting point in both scenarios consists in the application programmer providing a sketch of the parallel application structure, plus some ground annotations relative to the non-functional properties of interest. The ground annotation values may come from different sources:

execution times for business logic code from profiling; SLAs may directly come from the application programmer; times for execution of the pattern infrastructure will come from target architecture profiling.

In the first scenario, the application programmer may apply different rewriting rules through refactorers, ask for specific non-functional property values through visitors in a loop up to the point he/she is satisfied with the patterned application structure *and* with the predicted non-functional property values. At this point, the DSL code generator may be used to produce the actual application code and this may be tested on the target architecture. Depending on the results the application programmer gets from execution of the application on the target architecture, the process may be iterated again. This is needed as all the models we use in the DSL visitors and optimizers are approximated, theoretical models. Our experience is that these models give valuable qualitative results but not precise quantitative results. As a consequence, monitoring of exact values for non-functional properties on the target architecture will still be required for fine tuning of the application.

In the second scenario, the application programmer may simply apply one or more optimizers for the non-functional concern of interest and directly to the code generation phase. A fine tuning iterative process, as in the first scenario, may be necessary for the same reasons or–and in this case it could turn out to be a more coarse grained process–to find a trade-off while optimizing the application pattern structure when taking into account more than a single non-functional concern.

4 Experimental validation

In this Section we outline our experience with a *proof-of-concept* prototype of the DSP tool framework. We first briefly discuss the prototype implementation detail and then show results (qualitative and quantitative) achieved when designing patterned parallel applications with the prototype. Performance experiments have been conducted on a a Intel Ivy Bridge micro-architecture hosting 2 CPUs, each with 12 cores 2-way Hyper Threading.

4.1 Prototype implementation

The proof-of-concept prototype has been implemented using OCaml (`https://ocaml.org`), to take advantage of the symbolic computing facilities offered by ML-derived languages in general, and in particular to avail of OCaml's easy interface to standard Unix/Linux programming tools. Being a *proof-of-concept* prototype, we have not yet implemented the actual syntax shown in Sec. 2, but rather have employed a transitional DSL using the features provided by OCaml. A new version of the prototype is under development that uses Camlp5 (`http://camlp5.gforge.inria.fr/`) features to implement the actual DSL syntax. The prototype supports creation and use of abstract pattern terms as compositions of pipeline, farm, map, reduce, sequential composition and sequential code wrapper patterns. Defining a pattern expression is as simple as instantiating a Pattern abstract data type (see Fig. 4 (1)).

Annotations are provided through a derived data type and may be associated to any term of the pattern grammar as lists of elementary annotations. Elementary annotations are basically ground terms, as defined above. In the prototype we represent synthesized and inherited annotations as ground annotations for the sake of code simplicity (see Fig. 4 (2)).

We have visitors and optimizers implemented relative to the performance non-functional concern. For example, a visitor computes the service time of a pattern expression provided all the `Seq` nodes have some ServiceTime ground annotation (see Fig. 4 (3)).

```
(1) # let prog1 = Pipeline(Seq("f"), Seq("g"));;
    val prog1 : patterns = Pipe (Seq "f", Seq "g")
(2) # let prog2 = Apipe(Aseq("f",[Ts(2.0)]), Aseq("g",[Ts(4.0)]),[]);;
    val prog2 : annot_skel = Apipe (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), [])
(3) # servicetime prog2;;
    - : float = 4.
(4) # ts_optim prog2;;
    - : annot_skel =
    Apipe (Aseq ("f", [Ts 2.]), Afarm (Aseq ("g", [Ts 4.]), [Pd 2]), [])
    # normal_form prog2;;
    - : annot_skel = Afarm (Acomp (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), []), [])
(5) # pipe_elim prog1;;
    - : patterns = SeqComp (Seq "f", Seq "g")
    # farm_intro(pipe_elim prog1);;
    - : patterns = Farm (SeqComp (Seq "f", Seq "g"))
(6) # let appl = Afarm(Aseq("w",[Ts(5.0)]), [Pd(2); SLAts(1.0)]);;
    val appl : annot_skel = Afarm(Aseq("w",[Ts 5.]), [Pd 2; SLAts 1])
    # serviceetime appl;;
    - : float = 2.5
(7) # let appl_opt = ts_sla_optimizer appl;;
    appl_opt : annot_skel =  Afarm (Aseq ("w", [Ts 5.]), [Pd 5; SLAts 1.])
    # serviceetime appl_opt;;
    - : float = 1.
(8) # let appl = Afarm(Apipe(Aseq("stage1",[Ts 2.]), Aseq("stage2",[Ts 6.]),[]),[]);;
    - : annot_skel = Afarm(Apipe(Aseq("stage1",[Ts 2.]),Aseq ("stage2",[Ts 6.]),[]),[])
(9) # normal_form appl;;
    - : annot_skel = Afarm(Acomp(Aseq("stage1",[Ts 2.]),Aseq("stage2",[Ts 6.]),[]),[])
```

Figure 4: Prototype usage samples

A service time optimizer introducing suitable parallel patterns on "slow" pipeline stages as well as an optimizer computing the normal form of stream parallel patterns are also provided (see Fig. 4 (4)).

Furthermore, a refactorer implementing rewriting rules, including those listed in Table 2 is provided. Rewrite rules may be executed by invoking the refactored procedure named after the rewriting rule name (see Fig. 4 (5)).

4.2 Implementing an optimizer

We show how an optimizer may be implemented that ensures a user (application programmer) provided service time SLA in a stream parallel application.

As stated in Sec. 2.2.4, the application programmer may require that a given application guarantees a stated service time. We assume to start with the application pattern expression whose top level node is associated with the required SLA service time (see Fig. 4 (6)).

We define a visitor propagating the SLA down through the stream parallel pattern tree, according to the model stating that i) a pipeline SLA should be propagated unchanged to the pipeline stages, ii) a farm SLA requiring achievement of a service time t in an n_w worker farm should be propagated to the workers as an SLA requiring a $\frac{t}{n_w}$ service time in the case that the workers are not sequential patterns, and should not be propagated at all in the case that the workers are sequential stages. In our case, the application of the visitor to the `appl` term leaves

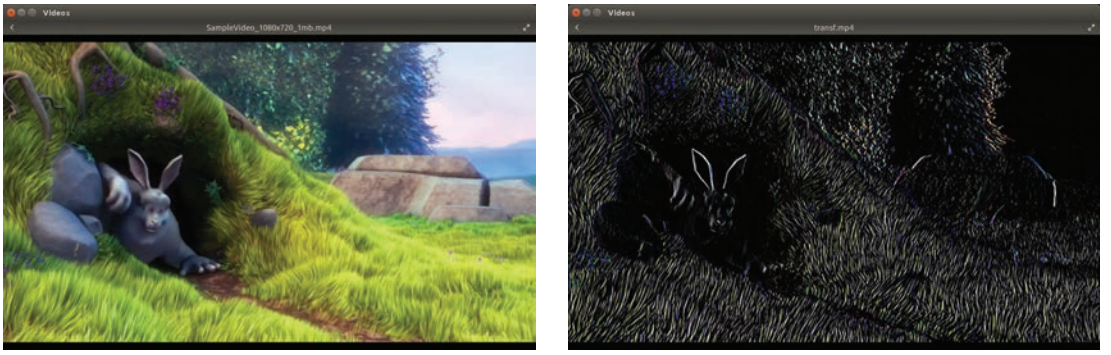

Figure 5: Original frame (left), transformed one (right)

the term unchanged. Finally, we define an optimizer where a bottom-up visitor assigns the correct number of workers to the farms (possibly transforming into farms those sub-trees that were not already parallel) in such a way that the service time required by the SLA is eventually achieved. By applying the optimizer we will therefore get the pattern expression in Fig. 4 (7), where the parallelism degree has been modified to accomplish the user supplied SLA.

4.3 Refactoring

We considered a video processing application that transforms each frame of the video by applying a couple of filters resulting in a kind of "emboss" effect. The effect is shown (on a single frame) in Fig. 5. The application is initially represented within our DSL tool as the pattern term: Pipeline(StreamSource(readVideo),Seq(filter1),Seq(filter2),StreamSink(writeFrame)) where the StreamSource(readVideo) and StreamSink(writeFrame) sequential wrappers read an MP4 file generating a stream of frames represented as OpenCV images, and write the OpenCV images appearing on a stream to an MP4 video file, respectively. The Seq(filter1) and Seq(filter2) implement a sharpener and an emboss filter, respectively. We know from profiling that the time spent computing the first and second filter is about 6 and 5.6 msec, respectively. We also know the StreamSource(readVideo) succeeds in outputting frames every 0.88 msecs. This may be modelled by using a pattern expression such as Pipeline(StreamSource(readVideo) with Ts(0.88 usec),Seq(filter1) with Ts(6 usec),Seq(filter2) with Ts(5.6 usec),StreamSink(writeFrame)) .

Pattern expression	T_C^{ideal}	T_C	Pd
P0: Pipeline(StreamSource(readVideo),Seq(filter1),Seq(filter2), StreamSink(writeFrame))	45.3 s	48.6 s	4
P1: Pipeline(StreamSource(readVideo),Farm(Seq(filter1)) with Pd(8), Farm(Seq(filter2)) with Pd(7),StreamSink(writeFrame))	6.7 s	8.2 s	21
P2: Pipeline(StreamSource(readVideo),Farm(SeqComp(Seq(filter1),Seq(filter2))) with Pd(15), StreamSink(writeFrame))	6.7 s	7.8 s	19
P3: FarmEWC(StreamSource(readVideo),SeqComp(Seq(filter1),Seq(filter2)), StreamSink(writeVideo)) with Pd(15)	6.7 s	7.7 s	17

Figure 6: Video processing application refactorings

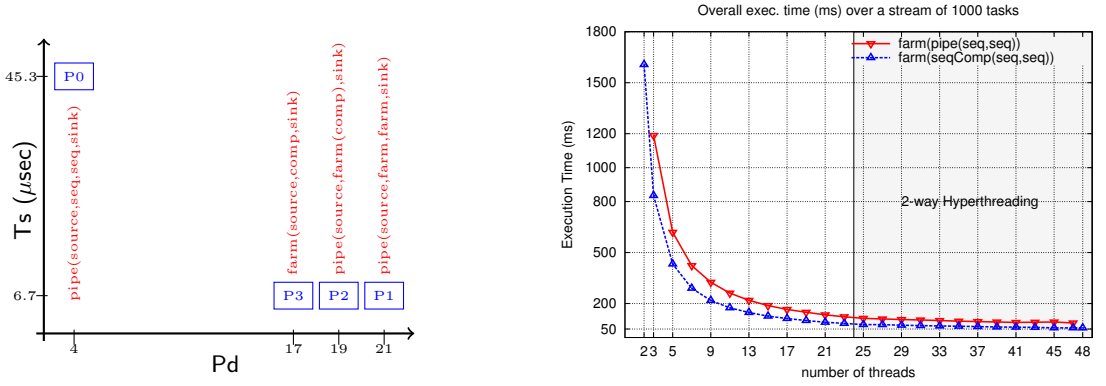

Figure 7: Video application pattern space (left), original Farm(Pipeline(Seq,Seq))) vs. normal form (Farm(SeqComp(Seq,Seq))) (right) performance.

Using the refactoring rules and the optimizers in our prototype implementation, we derived other pattern expressions equivalent to this one, namely those depicted in Fig. 6. In this case, we used a further rewriting rule which is specific to the FastFlow implementation of farm patterns (not listed in Fig. 2). The rule formalizes the possibility of collapsing a pipeline with a StreamSource, a Farm and a StreamSink into a single farm where the StreamSource and the StreamSink have been implemented in the farm emitter and collector service components[3] Pipeline(StreamSource(Pat_1),Farm(Pat_2),StreamSink(Pat_3) ≡ FarmEWC(Pat_1, Pat_2, Pat_3). The visitor computing the ideal completion time for the pattern expression[4] provides numbers (Fig. 6 2^{nd} col) that indicate that all the rewritings using farms are equivalent and are all much better (in terms of the expected completion time) than the first, pipeline only, expression. However, the visitor computing the parallelism degree (Fig. 6 4^{th} col) shows that there is a clear advantage in using the last row pattern expression, as it uses the minimum number of computing resources. As a consequence, the programmer will be presented with three refactorings whose performances are those sketched in Fig. 7 (left) and he/she will thus be steered toward employing refactoring P3. This hint is confirmed by the actual completion times achieved when processing an MP4 video with 7548 640x360 frames using OpenCV 2.4.9 to implement the filters and the stream source and sink, which are shown in Fig. 6 3^{rd} col. Refactoring P3 uses a smaller number of parallel executors and this turns out to be more efficient in terms of the added parallel resource management.

4.4 Normal form optimizer

We considered a program written by the application programmer as a farm whose workers are two-stage pipelines. Both pipeline stages are sequential. The latency of the second stage of the pipeline is three times the latency of the first stage (see Fig. 4 (8)).
Applying the normal form optimization we get the result shown in Fig. 4 (9)).

Fig. 7 (right) shows execution times achieved by the two versions of this application. The code generated is FastFlow code automatically produced from the original and from the normal form pattern expressions. The code has been compiled with the gcc 4.8.3 using the -O3 flag.

[3] in [5] the role of these service components of the FastFlow farm implementation is clearly exposed
[4] $T_C^{ideal}(m, T_S) = m \times T_S$ with T_S being the service time and m being the number of tasks to be processed

The maximum speedup of the original code is 23.3× while for the optimized normal form version it is 35.6×. It is worth pointing out that employing normal form, as with other rewriting rules considered in this work, does not affect the way data is distributed and propagated along the parallel activity graph *de facto* implementing the patterned parallel application. Composition of patterns is assumed to exploit the fact each pattern has a single source for the input data and a single destination for the computed result.

5 Related work

There is a rich literature of DSLs aimed at providing programmers with high-level parallel programming models. Delite [14] is a DSL platform providing building blocks for writing embedded high performance DSLs with low effort in different application domains. Building blocks are common components like parallel patterns and code generators that can be used in DSL implementations. Delite DSLs are embedded in the Scala language. It has been used for machine learning (OptiML) graph analysis (OptiGraph) and mesh-based PDE solvers (OptiMesh) [14].

Diderot [4] is a parallel DSL for image analysis demonstrating good performance compared to hand-written C code by using an optimized library. PolyMage [13] is a DSL and compiler for image processing pipelines expressed in a high-level declarative language. It provides a model-driven compiler that performs complex fusion, tiling, and storage optimization automatically. Obsidian [15] is an embedded DSL in Haskell providing the GPU programmer with a tool for design space exploration when implementing CUDA kernels.

There are several attempts to use DSLs to specify parallelism in C/C++ applications, for example SPar [11] for streaming computations and [16] for stencil computations. Nebo [7] is a declarative DSL embedded in C++ for PDEs targeting GPU based many-core platforms.

Algorithmic skeleton frameworks, such as SKePU [8], Muesli [9] and FastFlow [5] provide some capabilities for design space exploration. However, users have to write their applications using the framework syntax and they have to have sufficient knowledge of the framework details in order to obtain satisfactory performance. Finally, some works have considered effective means to explore a solution space that may be fully generated and then evaluated [1].

To the best of our knowledge, there are no other works aiming at providing a DSL to support the *design* phase of parallel applications built using only parallel design patterns.

6 Conclusions

In this work we have presented a DSL based toolchain supporting design of structured parallel applications where parallelism may be expressed and orchestrated only through the use of (compositions of) parallel design patterns. We first outlined the main features of our framework and then discussed a proof-of-concept implementation together with some qualitative and quantitative results obtained using this prototype. Results are encouraging, showing that the approach is feasible and that various known techniques may be framed in the tool in such a way that the parallel application programmer may be provided with a useful collection of tools accessed through a uniform and comprehensive DSL interface. The entire DSL design and implementation have been structured to be extensible. New patterns, rewriting rules, annotations, visitors and optimizers may be easily added to the framework while retaining all the possibilities offered by the original design and implementation.

Acknowledgements The DSL design and implementation have been partially supported by the EU project H2020-ICT-2014-1 no. 644235 *RePhrase* (http://rephrase-ict.eu).

References

[1] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 289–300, New York, NY, USA, 2014. ACM.

[2] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT*, pages 955–962. IASTED/ACTA press, 1999.

[3] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. Cost-directed refactoring for parallel erlang programs. *International Journal of Parallel Programming*, 42(4):564–582, 2014.

[4] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel dsl for image analysis and visualization. In *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 111–120, New York, NY, USA, 2012. ACM.

[5] Marco Danelutto and Massimo Torquati. Structured parallel programming with core fastflow. In Viktria Zsk, Zoltn Horvth, and Lehel Csat, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer International Publishing, 2015.

[6] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. State access patterns in embarrassingly parallel computations. In *Proc. of the HLPGPU 2016 Workshop, HiPEAC 2016, Prague*, 2016.

[7] Christopher Earl, Matthew Might, Abhishek Bagusetty, and James C. Sutherland. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. *Journal of Systems and Software*, (in press):–, 2016.

[8] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proc. of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[9] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *IJHPCN*, 7(2):129–138, 2012.

[10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, 2010.

[11] Dalvan Griebler and Luiz Gustavo Fernandes. Towards a domain-specific language for patterns-oriented parallel programming. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages*, volume 8129 of *LNCS*, pages 105–119. Springer Berlin Heidelberg, 2013.

[12] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[13] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[14] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[15] Bo Joel Svensson, Mary Sheeran, and Ryan Newton. Design exploration through code-generating dsls. *Queue*, 12(4):40:40–40:52, April 2014.

[16] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proc. of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.